Spring 2019 CS 1110 Final Exam Solutions

Please turn off and stow away all electronic devices. You may not use them for any reason during the exam. Do not bring them with you if you leave the room temporarily.

This is a closed book and notes examination. You may use the **2-sided reference sheet at the back of the exam.**

There are **7 problems**. Make sure you have the whole exam.

You have **150 minutes** to complete 120 points. Use your time accordingly.

Question	Points	Score
1	9	
2	16	
3	20	
4	14	
5	22	
6	22	
7	17	
Total:	120	

It is a violation of the Academic Integrity Code to look at any exam other than your own, to look at any other reference material, or to otherwise give or receive unauthorized help.

We also ask that you not discuss this exam with students who are scheduled to take a later makeup.

Academic Integrity is expected of all students of Cornell University at all times, whether in the presence or absence of members of the faculty. Understanding this, I declare I shall not give, use or receive unauthorized aid in this examination.

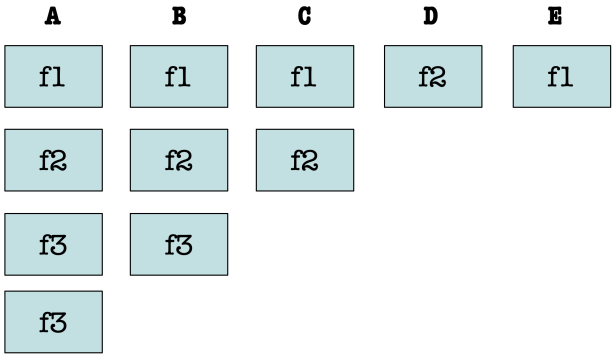
Signature: _____ Date _____

Name: _____ NetID _____

1. **Try, Try Again**

(a) [3 points] If Python has just finished printing `"SMILE!"` for the **5th time** and done nothing more, what does the call stack look like?

```
1  def f3():
2      print("SMILE!")
3      print("SMILE!")
4
5  def f2():
6      print("SMILE!")
7      f3()
8      print("SMILE!")
9      f3()
10     f3()
11
12 def f1():
13     print("SMILE!")
14     f2()
15
16 f1()
```



Correct Answer: C

(b) [3 points] Suppose that `fun1` is a **class method** for the class `C` and it has the following line of code in it:

```
a1 = b1 * 2
```

Where might the variable `b1` that was referred to in this line of code be located?

A. the global space
B. the call frame for `fun1`
C. the call frame of the function that called `fun1`
D. an instance/object attribute of an object of type `C`
E. a class attribute of class `C`

List all that apply: A,B

(c) [3 points] Consider a `Person` class with attributes `children` (a list of children) and `n_male` and `n_female` with the class invariant: `n_male + n_female == len(children)`
Think about how one would implement the class method `add_child(self, child, is_male)`. What is true of this invariant?

A. If the invariant is ever not true, Python will throw an error.
B. It must be true after every line of `add_child` executes.
C. It must be true before and after `add_child` executes.
D. A and B
E. B and C
F. A and C
G. A, B, and C

Correct Answer: C

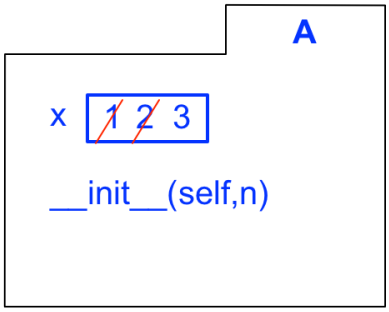
2. [16 points] **Keep it Classy**. Use the diagram on the right to show the state of Global Space, Class Folders, and Object folders after the code below finishes executing. You do **not** need to draw the call frames. This code runs without error.

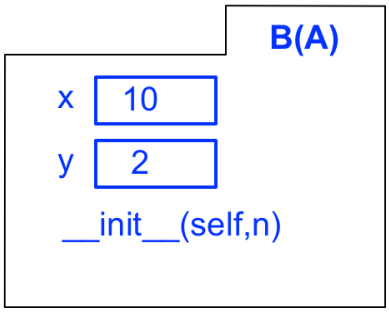
```
1  class A():
2      x = 1
3
4      def __init__(self, n):
5          self.y = n
6          A.x += 1
7
8  class B(A):
9      x = 10
10     y = 2
11
12     def __init__(self, n):
13         sum = self.y
14         super().__init__(n)
15         sum += self.y
16         self.y = sum
17         self.x = n
18
19 a = A(3)
20 b = B(5)
```

Global Space

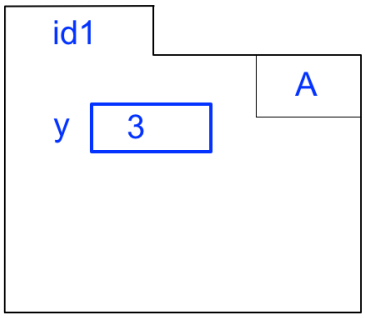
you are welcome to draw the call frames, but they will not be graded nor will graders look at them to give partial credit.

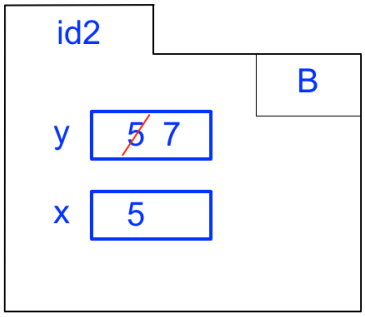
Class Folders





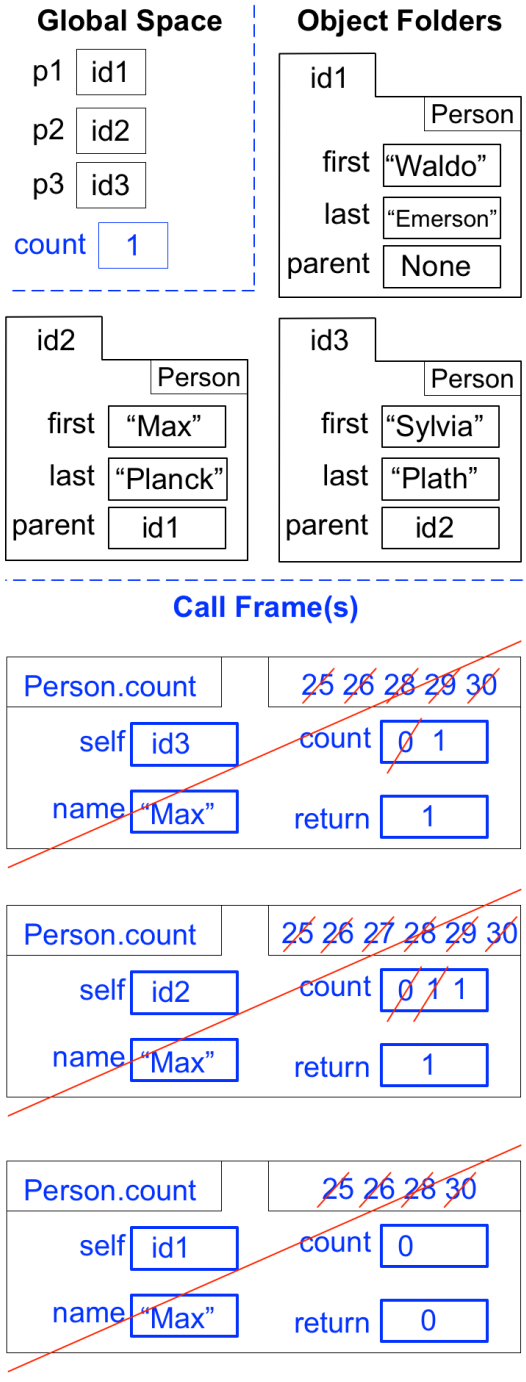
Object Folders





3. [20 points] **Another way to find Max.** The drawing below shows the state of memory **after executing lines 1-34** of the code, ignoring class folders for simplicity. **Update the drawing, adding any call frame(s)** or changes resulting from executing `line 35`. If you cross out a value or call frame, make sure it is still legible.

```
1  class Person():
2      """ A class representing a person
3      in a 1-parent world."""
4
5      def __init__(self,first,last,parent):
6          """
7          Creates a new Person with 3
8          instance attributes.
9
10         first: non-empty str of letters
11         last:  non-empty str of letters
12         parent: a Person or None
13         """
14         self.first = first
15         self.last = last
16         self.parent = parent
17
18     def count(self, name):
19         """
20         Counts ancestors (incl. self) with
21         first name matching parameter name.
22
23         Returns: an integer
24         """
25         count = 0
26         if self.first == name:
27             count = 1
28         if self.parent != None:
29             count += self.parent.count(name)
30         return count
31
32 p1 = Person("Waldo", "Emerson", None)
33 p2 = Person("Max", "Planck", p1)
34 p3 = Person("Sylvia", "Plath", p2)
35 count = p3.count("Max") # EXECUTE THIS!
```



4. [14 points] **More than a `Person`.** The definition of the `Person` class from the previous question is copied here for your convenience. **Define a subclass of `Person` called `Student`.** A `Student` has the attributes of a `Person` plus one additional attribute, `netID`.

```
class Person():
    """ A class representing a person
    in a 1-parent world."""

    def __init__(self,first,last,parent):
        """
        Creates a new Person with 3
        instance attributes.

        first: non-empty str of letters
        last:  non-empty str of letters
        parent: a Person or None
        """
        self.first = first
        self.last = last
        self.parent = parent
```

`netID` is **not** a parameter to any `__init__` method; it is a string that is created at initialization by concatenating 3 things:

1. the *lower-case* first letter of the **first** name
2. the *lower-case* first letter of the **last** name
3. a unique number **across all students** representing when the `Student` was created. (Note: this is a simplification of how Cornell actually assigns your `netID` a number.)

Examples:
The very first student at Cornell, Ezra Cornell, has `netID "ec1"`.
The second student at Cornell, Pearl Buck, has `netID "pb2"`.
The third student at Cornell, Martha Pollack, has `netID "mp3"`.

Your subclass should make use of the `Person` class functionality and avoid code redundancy. Do **not** worry about enforcing preconditions, writing comments, or docstrings.

```
class Student(Person):
    count = 0

    def __init__(self,first,last,parent):
        super().__init__(first,last,parent)
        Student.count += 1
        self.netID = first[0].lower()+last[0].lower()+str(Student.count)
```

5. **The Sorted Hat.** In this question you will consider two approaches to implementing the function `is_sorted`, which determines whether a list of integers `b` is sorted (in ascending order) or not. In *neither* part are you responsible for asserting/enforcing preconditions.

   (a) [12 points] **While Loops and Loop Invariants.** This version is implemented using a while loop. You are given the precondition, postcondition, the loop invariant, and the structure of the while loop. You must provide the initialization, the loop condition, and the loop body.

```
def is_sorted(b):
    """
    Returns: True if b is sorted in ascending order, False otherwise

    b: a list of integers with at least 1 element; remains unchanged

    Examples:
    is_sorted([3]) Returns True
    is_sorted([3,3]) Returns True
    is_sorted([3,4]) Returns True
    is_sorted([-4,1,-12]) Returns False
    """
    # PRE: b is a list of integers with at least 1 element
    # TASK #1: initialize these 2 variables so that the loop
    # invariant is true at the start

    sorted_this_far = True
    k = 0

    # INV: sorted_this_far is True if b[0..k] is sorted, otherwise False
    # TASK #2: provide the loop condition so that the loop
    # terminates as soon as it knows the list is not sorted
    # Also, make sure you do not inspect past the end of the list

    while (sorted_this_far and k < len(b)-1):

        # TASK #3: provide the loop body

        sorted_this_far = b[k] <= b[k+1]

        k = k + 1

    # POST: sorted_this_far is True if b is sorted, otherwise False
    # TASK #4: what should this function return?

    return sorted_this_far
```

(b) [10 points] **Recursion.** Make effective use of **recursion** to provide a second implementation of `is_sorted`. Your solution *must use recursion* in order to receive points. When you have finished, step through your code to make sure it works on the given examples. The spec has been copied for your convenience.

```
def is_sorted(b):
    """
    Returns: True if b is sorted in ascending order, False otherwise

    b: a list of integers with at least 1 element; remains unchanged

    Examples:
    is_sorted([3]) Returns True
    is_sorted([3,3]) Returns True
    is_sorted([3,4]) Returns True
    is_sorted([-4,1,-12]) Returns False
    """

    if len(b) == 1:
        return True
    return b[0] <= b[1] and is_sorted(b[1:])
```

6. **Shop till you drop!** For this question, you will answer questions about and also help complete a new class called `Product`.

```
class Product():
    """An instance represents an item that can be sold. """
    SALES_TAX_RATE = 0.04

    def __init__(self, name, price, quantity, tax_exempt):
        """A new product item called "name" with 4 attributes:
        name:       a non-empty str, e.g., 'Milk'
        price:      a float > 0.0
        quantity:   a non-negative (but possibly 0) int indicating
                    how many of these items are in stock
        tax_exempt: a bool indicating whether sales tax is added
                    to the purchase price of this item or not
        """

        assert type(name) == str
        assert len(name) > 0
        self.name = name
        assert type(price) == float
        assert price > 0.0
        self.price = price
        assert type(quantity) == int
        assert quantity >= 0
        self.quantity = quantity
        assert type(tax_exempt) == bool
        self.tax_exempt = tax_exempt
```

(a) [10 points] Complete the `__init__` method above according to its specification. Be sure to assert all of the stated preconditions.

(b) [2 points] Why is the attribute `SALES_TAX_RATE` in all caps?
Answer in 1 sentence and be succinct. Irrelevant statements will cost you points.

All caps indicates that this attribute's value should be considered a constant and never be changed.

(c) [4 points] This page continues the `Product` class definition from the previous page. Complete the `__str__` method below according to its specification.

```
def __str__(self):
    """
    Returns: a [str] representation of the Product, including all
    4 attributes separated by commas, in a string form. The price
    should have a dollar sign. Don't worry about extending the
    price to exactly 2 decimal places.
    Example: "Milk, $3.0, 10, True"
    """

    return self.name+", $"+str(self.price)+", "+\
        str(self.quantity)+", "+str(self.tax_exempt)
```

(d) [4 points] Complete the `__eq__` method below according to its specification.

```
def __eq__(self, other):
    """Returns: True if other is a Product and both self and other
    have the same name, price, and tax-exempt status. False otherwise.
    """

    return isinstance(other,Product) and \
        self.name == other.name and \
        self.price == other.price and \
        self.tax_exempt == other.tax_exempt;
```

(e) [2 points] You aren't sure whether your `__eq__` method is being called or not. Maybe you gave it the wrong name? Maybe the underscores are wrong? You're not sure. Explain how you could modify `__eq__` above so that you could find out whether `__eq__` ever gets called. Your solution should work without modifying any other aspects of the `Product` class.

Place a print statement in the first line of the body of the method. Something like `print("eq was called!")`. If the print statement never appears you know your method is not getting called. If you see it appear on the screen you know it is.

7. **What's in store for you?** Do not start this question until you have given the previous question a serious attempt. This question introduces a `Store` class; `Store`s contain `Product`s.

   (a) [12 points] **List version.** Complete the `stock` method below according to its specification. You do **not** need to assert any preconditions.

```
class Store():
    """An instance represents a named store with goods to sell

    INSTANCE ATTRIBUTES:
    name:  the name of the store [str], Example: "Aldi"
    goods: a list of Product that the store has in stock """

    def __init__(self, name):
        """Creates a new store called "name" with 2 attributes:
        name:      a non-empty str, e.g. 'Aldi'
        goods:     a (possibly empty) list of Product """
```

Implementation left out; you do not need to complete it

```
    def stock(self, p):
        """ p:  a Product to be added to the store

        - If p is NOT already in the store, add p to the
            store's goods.
        - If p IS already there, increase the store's products's
        inventory (quantity) by the quantity in the parameter p.

        Using "in" to test if p is already on the list WILL NOT WORK.
        Instead, check each element in goods for equality with p,
        making use of the equals method you wrote for Product.  """

        making use of the equals method you wrote for Product.  """
        found = False
        for g in self.goods:
            if g == p:
                g.quantity += p.quantity
                found = True
                break # speeds it up; not necessary for correctness
        if not found:
            self.goods.append(p)
```

(b) [5 points] **Dictionary version.** Did you notice on the previous page how tedious it was to check every element in the store to see whether a particular product was present? This page presents an **alternate** definition of `Store` in which `goods` is a *dictionary*, not a list. Re-implement the `stock` method below. The specification has changed slightly because `goods` is now a *dictionary*, not a list. You do **not** need to assert any preconditions. You may find the Dictionary Operations on the Reference Sheet helpful.

```
class Store():
    """An instance represents a named store with goods to sell

    INSTANCE ATTRIBUTES:
    name:  the name of the store [str], Example: "Aldi"
    goods: a dictionary keeping track of inventory.
           Each key is a str (the name of the Product). Note: we can
              no longer keep inventory for two Products that have the
              same name but different prices or tax-exemption status.
           Each value is a Product.

    Example:
          s = Store("Aldi")
          p1 = Product("Milk", 3.0, 10, True);
          s.goods[p1.name] = p1  # overwrites any existing Product w/ same name
          s.goods["Milk"] = p1   # alternative to the previous line """
```

`init` *implementation omitted for space. You do not need to complete it.*

```
    def stock(self, p):
        """ p:  a Product to be added to the store

        - If a Product with the same name as p is NOT already in the
            store, add p to the store's goods.
        - If a Product with the same name as p IS already there, ignore
            any price/exemption difference; increase the store's products's
            inventory (quantity) by the quantity in the parameter p.
        Using "in" to test if p is already in the dictionary WORKS! """

        if p.name in self.goods:
            v = self.goods[p.name]
            v.quantity += p.quantity
        else:
            self.goods[p.name] = p
```